SMART

Stochastic Model Analyzer for Reliability and Timing

G. Ciardo[1] R. L. Jones, III[1] A. S. Miner[2] R. Siminiceanu[1]

[1] Department of Computer Science, College of William and Mary
[2] Department of Computer Science, Iowa State University

SMART is a software package that integrates various logic and stochastic modeling formalisms into a single environment. Models expressed in different formalisms can be combined in the same study. For the analysis of the logical behavior, both explicit and symbolic state-space generation techniques, as well as CTL model checking algorithms, are available. For the study of the stochastic and timing behavior, both explicit and Kronecker numerical solution approaches are available, in addition to simulation. Since SMART is intended as an industry tool and a research tool, it is written in a modular way that allows for easy integration of new formalisms and solution algorithms.

1 SMART Language

SMART uses a strongly-typed declarative language with the following four basic predefined *types*:

- `bool`: `true` or `false`. `bool c := 3 - 2 > 0;`
- `int`: integer values (machine-dependent range). `int i := 12;`
- `real`: floating-point values (machine-dependent range and precision). `real x := sqrt(2.3);`
- `string`: character-array values. `string s := "Monday";`

Composite types can be defined using the concepts of:

- *sets*: collection of homogeneous objects. `{1,2,4,8,16}`
- *arrays*: multidimensional collections of homogeneous objects indexed by set elements. `a[3][0.2]`
- *aggregates*: analogous to the Pascal "record". `p:3`

A type can be further modified by the following *natures*, which describe stochastic characteristics:

- `const`: (the default) a non-stochastic quantity.
- `ph`: a random variable with discrete or continuous phase-type distribution.
- `rand`: a random variable with arbitrary distribution.
- `ctmc`, `dtmc`, `spn`, ...: stochastic formalisms defining a stochastic process indexed by time.

1.1 Function declarations

Syntactically, objects defined in SMART are functions, possibly recursive, and can be overloaded:

```
real pi := 3.14;                                      /* a parameter-less function */
bool close(real a, real b) := abs(a-b) < 0.00001;   /* a two-parameter function */
int pow(int base, int exp) := cond(exp==1,base,base*pow(base,exp-1));
real pow(real base, int exp) := cond(exp==1,base,base*pow(base,exp-1));
pow(5,3);                                             /* returns 125, integer */
pow(5.0,3);                                           /* returns 125.0, real */
```

1.2 Arrays

Arrays are declared using a `for` statement; their dimensionality is determined by the enclosing iterators. Since the indices along each dimension belong to a finite set, we can define arrays with `real` indices:

```
for (int i in {1..5}, real r in {1..i..0.1}) {
  real res[i][r]:= MyModel(i,r).out1;
}
```

fills array `res` with the value of measure `out1` for `MyModel`, when the first input parameter ranges from one to five and the second one ranges from one to the value of the first parameter, with a step of 1/10.

1.3 Fixed-point iterations

The approximate solution of a model is often based on a heuristic decomposition, where (sub)models are solved in a fixed-point iteration. This can be specified with the `converge` statement:

```
converge {
  real x guess 1.0;
  real y := fy(x, y);
  real x := fx(x, y);
}
```

The iterations stop when two subsequent `x` and `y` values differ by less than ϵ in either relative or absolute terms. The values for `x` and `y` are updated either immediately or at the end of each iteration. Both ϵ and the updating criterion are fine-tuned using *option* statements.

The `converge` and `for` statements can be arbitrarily nested within each other.

2 Random variables

SMART can manipulate discrete and continuous phase-type distributions. Combining `ph` types produces another `ph` type if phase-type distributions are closed under that operation:

```
ph int  X := geometric(0.7);
ph int  Y := equilikely(1,5);
ph int  A := min(3*X, Y);
ph int  B := 3*X+Y;
ph int  C := choose(0.4:3*X, 0.6:Y);
ph real x := expo(3.2);
ph real y := erlang(4,5);
ph real a := min(3*x, y);
```

However, mixing `ph int` and `ph real`, or performing other operations not guaranteed to result in a phase-type distribution, forces SMART to consider the random variable as generally distributed:

```
rand int  D := X-Y;
rand int  F := X*Y;
rand real E := x+X;
```

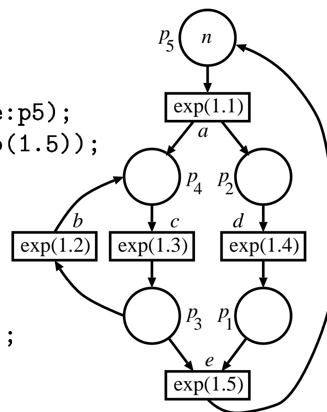
A `rand` random variable can be manipulated only via Monte Carlo methods (under development).

3 Model formalisms

A model is declared just like a function except that, instead of a return value, it specifies a block containing *declarations*, *specifications*, and *measures*. Components of the model are declared using formalism-specific types (e.g., the places of a Petri net). The model structure is specified by using formalism-specific functions (e.g., the arcs of a Petri net). Measures are user-defined functions that specify some quantity of interest (e.g., the expected number of tokens in a given place in steady-state), and are the only model components that can be accessed outside of the model definition block.

The design of SMART allows for relatively easy addition of new model formalisms. Currently, the `dtmc` (discrete-time Markov chains), `ctmc` (continuous-time Markov chains), and `spn` (stochastic Petri nets) formalisms are implemented. For the `spn` formalism, the type of underlying stochastic process is determined by the distributions specified for the transitions. For example, the model shown on the right is defined as:

```
spn mynet(int n) := {
  place p5, p4, p3, p2, p1;
  trans a, b, c, d, e;
  arcs(p5:a,a:p4,a:p2,p4:c,c:p3,p3:b,b:p4,p2:d,d:p1,p1:e,p3:e,e:p5);
  firing(a:expo(1.1),b:expo(1.2),c:expo(1.3),d:expo(1.4),e:expo(1.5));
  init(p5:n);
  int count := Count(false);
  real speed := ssavg(rate(a));
};
for (int n in {1..4}) {
  print("When n=",n," the number of states is ",mynet(n).count);
  print(" and the throughput is ",mynet(n).speed,"\n");
}
```



where `place` and `trans` are formalism-specific types; `arcs`, `firing`, and `init` are formalism-specific functions; `count` and `speed` are two measures. The `for` loop outside the model produces the output

```
When n=1 the number of states is 5 and the throughput is 0.284926
When n=2 the number of states is 14 and the throughput is 0.456624
When n=3 the number of states is 30 and the throughput is 0.553202
When n=4 the number of states is 55 and the throughput is 0.612783
```

where the measure calls cause SMART to perform the appropriate analysis.

Arrays and `for` loops are allowed within a model in the usual manner.

4 Advanced features

SMART implements various state-of-the-art solution algorithms that can be used for logical and stochastic analysis of large and complex models.

4.1 State-space generation and storage

Since the generation and storage of the state space is a major component of any state-space based solution technique, and it is a major component in model checking, we have investigated and implemented several strategies that can achieve much better time and memory efficiency than the simplistic explicit approach traditionally employed by "old generation" tools, where each state is stored as a separate object in its entirety. Using *multi-valued decision diagrams (MDDs)*, we are able to generate extremely large state spaces in a matter of seconds, minutes, or at most hours, depending on the model structure [2, 3, 7]. The famous model of dining philosophers is a particularly good example, as is shown in the table below. Note that using a technique called "saturation" [3], SMART requires very little memory and CPU time to generate extremely large state spaces for this model. When the model is Markovian, using MDDs for generation and storage of the state space $\mathcal{S}$ also allows for efficient state search and index computation, operations required for the numerical solution of the underlying process.

Number of Philosophers	States $\lvert\mathcal{S}\rvert$	MDD Nodes		Memory (bytes)		CPU (secs)
		Final	Peak	Final	Peak	
100	4.97×10^{62}	197	246	30,732	38,376	0.04
300	1.23×10^{188}	597	746	93,132	116,376	0.13
1,000	9.18×10^{626}	1,997	2,496	311,532	389,376	0.45
3,000	7.74×10^{1880}	5,997	7,496	935,532	1,169,376	1.34

MDD techniques require a decomposition of the overall model into submodels. For `spn` models, this is accomplished by partitioning their places. For example, the statement `partition(p5,p4,p3,p2,p1);` specifies that each place is in a submodel by itself, while `partition(p5,p4:p3,p2:p1);` specifies three submodels: `p5` and the transitions connected to it, `p4` and `p3` and the transitions connected to them, and `p2` and `p1` and the transitions connected to them. Note that multiple submodels may share a transition. The overall state space $\mathcal{S}$ of a partitioned model can be seen as a subset of the cross-product $\mathcal{S}_K \times \cdots \times \mathcal{S}_1$ of the *local state spaces* of the K submodels.

4.2 CTL model checking

CTL model checking queries are available in SMART via a set of model-dependent measures. The answers to CTL queries are logically stored as sets of states (the states that satisfy the formula), and physically as MDD trees with shared nodes. A data type `stateset` is reserved for the above structure. All the algorithms employ symbolic techniques, therefore a user-defined partitioning is required. There are four categories of functions:

- **atom builders:** `nostates` (empty set), `initialstate`, `potential` (states satisfying a condition);
- **set operators:** `union`, `intersection`, `complement`, `difference`, `includes`, `equal`, `neq`;
- **temporal logic operators:** `next` and `prev` (one-step reachability), `forward` and `backward` (reachability), plus the CTL operators `eg`, `eu`, `ag`, `au`;
- **utility functions:** to count and list states.

The following example shows one possible way to check for absorbing (deadlocked) states in a model, then verify some stability and safety properties:

```
stateset U         := forward(initialstate);   /* reachable states */
stateset NotAbsorb := prev(potential(true));   /* states that have a successor */
stateset Absorb    := difference(U,NotAbsorb); /* reachable absorbing states */
bool     deadlock  := neq(Absorb,nostates);    /* deadlock detection */
int      prnabs    := printset(Absorb);        /* list deadlocked states */
stateset Good      := potential(expr1);
stateset Bad       := potential(expr2);        /* before faulting, the system ...
stateset Safe      := au(Bad,Good);            /* ... is always in a good state */
stateset Stable    := eg(Good);                /* there is an infinite good run */
```

4.3 Markov and non-Markov models

Stochastic Petri nets (`spn` models) having the most convenient underlying stochastic processes to study include those with `geometric` firing delays, which have an underlying `dtmc`, and those with `expo` firing delays, which have an underlying `ctmc`. In either case, SMART provides standard numerical solutions to these Markovian models: power method and uniformization as appropriate for transient analysis, and iterative methods (Jacobi, Gauss-Seidel, SOR) for steady-state analysis.

When `ph int` (resp. `ph real`) transitions are used alone (possibly in conjunction with "immediate transitions" having zero time delay), an otherwise non-Markovian process becomes a `dtmc` (resp. `ctmc`) on an expanded state space that encodes the distribution of firing delays along with each reachable marking. However, *mixing* `ph int` and `ph real` transitions within the *same* `spn` model complicates matters in ways that

may require study by simulation, but not necessarily. A "mixed" `ph` model may also enjoy Markovian analysis if active `ph int` transitions are *synchronized* in the sense that they share a common clock, which records the time since the last phase advancement. With the remaining firing time distributions already encoded in the state, the resulting stochastic process is semi-regenerative. As such, a single *embedded* `dtmc` model arises separately from but interacting with many `ctmc` models that arise from `ph real` transition firings, all of which can be studied using standard methods. SMART currently provides numerical steady-state analysis of mixed `ph` models that have underlying semi-regenerative processes, which is automatically verified while generating the state space. See [6] for a more formal and detailed treatment of this subject, including the subtleties of marking dependencies, preemption, (age) memory policies, and the solution algorithm itself.

Of course, simulation could also be used to study the models mentioned above and is currently the only way to study models that do not admit semi-regenerative processes: those with *asynchronous* `ph int` transitions. However, we are investigating numerical methods capable of studying such models even if only approximately. In addition to the standard solution methods mentioned above, we are also exploring multigrid-like methods, distributed solutions, and the automatic computation of optimal relaxation parameters.

4.4 Kronecker encoding of the Markov chain matrix

SMART provides solution methods based on a Kronecker expression of the transition rate matrix $\mathbf{R}$ of the CTMC underlying a continuous Markov model. These methods are quite effective in reducing the memory requirements. Just like MDD methods, they require a user-specified partition of the model into submodels. Indeed, both MDD and Kronecker techniques can be employed for the solution process, to store $\mathcal{S}$ and $\mathbf{R}$, respectively. This allows SMART to use probability vectors of size $|\mathcal{S}|$, which are then the only objects for which the memory requirements effectively limit the applicability of a numerical solution technique [1, 5].

SMART also offers a particularly efficient data structure that combines the idea of decision diagrams with that of Kronecker algebra: *matrix diagrams* [4]. When this option is exercised, the computational overhead required for Kronecker-based approaches is significantly lower. For example, the following table shows the memory and runtime results for the model of a Kanban system [9], parameterized by the number of tokens n initially in each of the $K = 4$ submodels in which it is decomposed. Clearly, the number of states $|\mathcal{S}|$ and the number of nonzero entries $\eta(\mathbf{R})$ grow very rapidly, but, with the Kronecker option, SMART can solve models having over one order of magnitude more states than with the traditional approach. Furthermore, from the columns showing the time required (seconds per iteration), we can conclude that the time overhead imposed by the Kronecker approach is smallest when using matrix diagrams.

n	$\lvert\mathcal{S}\rvert$	$\eta(\mathbf{R})$	Matrix diagram			Kronecker					Ordinary	
			Memory	Gauss-Seidel		Memory	Gauss-Seidel		JOR ($\omega = 0.9$)		Gauss-Seidel	
			(bytes)	Iters	sec/iter	(bytes)	Iters	sec/iter	Iters	sec/iter	Iters	sec/iter
4	454,475	3,979,850	13,518	99	12.33	6,096	149	23.69	370	11.99	149	3.04
5	2,546,432	24,460,016	21,667	139	73.09	9,486	214	147.70	527	74.09	214	18.51
6	11,261,376	115,708,992	32,702	185	336.21	14,106	289	723.30	713	359.15	-	-
7	41,644,800	450,455,040	46,678	238	1,289.91	20,388	374	2,922.80	-	-	-	-

4.5 Approximations

The use of MDDs for $\mathcal{S}$ and a Kronecker encoding for $\mathbf{R}$ allow us to represent the underlying CTMC extremely efficiently. For numerical solution, the only remaining memory bottleneck is the solution vector. This allows SMART to compute exact performance measures for large systems. When the solution vector is too large (due to an extremely large state space), other techniques, such as discrete-event simulation or approximation techniques, must be used.

Currently, SMART provides a novel approximation technique [8] that works for stationary analysis of models with an underlying CTMC. As far as we know, it is the only approximation technique that makes use of complete knowledge of $\mathcal{S}$ and $\mathbf{R}$. The technique performs K approximate aggregations, where each aggregation is based on the structure of the MDD representing $\mathcal{S}$. Since each aggregation may depend on the probabilities computed for the other aggregations, fixed-point iterations are used to break cyclic dependencies. The results can be quite accurate, as shown in the table below for the model of a Kanban system. The table shows the worst relative error (when an exact comparison was possible) when computing

the average number of tokens in each place, and the average firing rate of each transition. The total CPU time required by the approximation is also shown.

N	$\|S\|$	Worst relative error		CPU
		Avg. #Tokens	Trans. Rate	(sec)
4	4.54×10^5	2.846%	-0.016%	0.47
5	2.55×10^6	2.557%	-0.074%	0.84
6	1.13×10^7	2.262%	-0.099%	1.38
7	4.16×10^7	2.032%	-0.097%	2.19
30	4.99×10^{13}	—	—	462.48
66	1.99×10^{17}	—	—	13,424.50

References

[1] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper. Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models. *INFORMS J. Comp.*, 12(3):203–222, 2000.

[2] G. Ciardo, G. Luettgen, and R. Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In M. Nielsen and D. Simpson, editors, *Application and Theory of Petri Nets 2000 (Proc. 21th Int. Conf. on Applications and Theory of Petri Nets, Aarhus, Denmark)*, Lecture Notes in Computer Science 1825, pages 103–122. Springer-Verlag, June 2000.

[3] G. Ciardo, G. Luettgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In T. Margaria and W. Yi, editors, *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science 2031, pages 328–342, Genova, Italy, Apr. 2001. Springer-Verlag.

[4] G. Ciardo and A. S. Miner. A data structure for the efficient Kronecker solution of GSPNs. In P. Buchholz, editor, *Proc. 8th Int. Workshop on Petri Nets and Performance Models (PNPM'99)*, pages 22–31, Zaragoza, Spain, Sept. 1999. IEEE Comp. Soc. Press.

[5] G. Ciardo and M. Tilgner. On the use of Kronecker operators for the solution of generalized stochastic Petri nets. ICASE Report 96-35, Institute for Computer Applications in Science and Engineering, Hampton, VA, May 1996.

[6] R. L. Jones. Analysis of Phase-Type Stochastic Petri Nets with Discrete and Continuous Timing. Technical Report CR-2000-210296, NASA Langley Research Center, Hampton, VA, June 2000.

[7] A. S. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In H. Kleijn and S. Donatelli, editors, *Application and Theory of Petri Nets 1999 (Proc. 20th Int. Conf. on Applications and Theory of Petri Nets, Williamsburg, VA, USA)*, Lecture Notes in Computer Science 1639, pages 6–25. Springer-Verlag, June 1999.

[8] A. S. Miner, G. Ciardo, and S. Donatelli. Using the exact state space of a Markov model to compute approximate stationary measures. In J. Kurose and P. Nain, editors, *Proc. 2000 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 207–216, Santa Clara, CA, June 2000. ACM Press.

[9] M. Tilgner, Y. Takahashi, and G. Ciardo. SNS 1.0: Synchronized Network Solver. In *1st International Workshop on Manufacturing and Petri Nets*, pages 215–234, Osaka, Japan, June 1996.